Chapter 18

Introduction to Pointers

In This Chapter

- Using the `sizeof` operator
- Grabbing a variable's memory location
- Creating pointer variables
- Using a pointer to peek at data
- Assigning values by using pointers

It's considered one of the most frightening topics in all of programming. *Boo!*

Pointers scare a lot of beginning C programmers — and even experienced programmers of other languages. I believe that the reason for the fear and misunderstanding is that no one bothers to explain in fun, scintillating detail how pointers really work. So clear your mind, crack your knuckles, and get ready to embrace one of the C language's most unique and powerful features.

The Biggest Problem with Pointers

It's true that you can program in C and avoid pointers. I did it for a long time when I began to learn C programming. Array notation offers a quick-and-dirty work-around for pointers, and you can fake your way around the various pointer functions, hoping that you get it right. But that's not why you bought this book!

After working with pointers for some time and understanding the grief they cause, I've come up with a reason for the fear and dread they induce: Pointers are misnamed.

I can reason why a pointer is called a pointer: It points at something, a location in memory. The problem with this description is that most pedants explain how a pointer works by uttering the phrase, "A pointer points. . . ." That's just wrong. It confuses the issue.

Adding to the name confusion is the fact that pointers have two personalities. One side is a variable that holds a memory location, an *address.* The other side reveals the value at that address. In that way, the pointer should be called a *peeker.* This chapter helps straighten out the confusion.

- The pointer is a part of the C programming language that's considered low-level. It gives you direct access to computer memory, information that other languages — and even operating systems — prefer that you not touch. For that reason:
- A pointer can get you into trouble faster than any other part of C programming. Be prepared to witness memory segmentation errors, bus errors, core dumps, and all sorts of havoc as you experiment with, and begin to understand, pointers.

Sizing Up Variable Storage

Digital storage is measured in bytes. All the information stored inside memory is simply a mass of data, bits piled upon bits, bytes upon bytes. It's up to the software to make sense of all that.

Understanding variable storage

In C, data is categorized by storage type (`char`, `int`, `float`, or `double`) and further classified by keyword (`long`, `short`, `signed`, or `unsigned`). Despite the chaos inside memory, your program's storage is organized into these values, ready for use in your code.

Inside a running program, a variable is described by these attributes:

- Name
- Type
- Contents
- Location

The *name* is the name you give the variable. The name is used only in your code, not when the program runs.

The *type* is one of the C language's variable types: `char`, `int`, `float`, and `double`.

The *contents* are set in your program when a variable is assigned a value. Though data at the variable's storage location may exist beforehand, it's considered garbage, and the variable is considered uninitialized until it's assigned a value.

The *location* is an address, a spot inside the device's memory. This aspect of a variable is something you don't need to dictate; the program and operating system negotiate where information is stored internally. When the program runs, it uses the location to access a variable's data.

Of these aspects, the variable's name, type, and contents are already known to you. The variable's location can also be gathered. Not only that, but the location can be manipulated, which is the inspiration behind pointers.

Reading a variable's size

How big is a `char`? How long is a `long`? You can look up these definitions in Appendix D, but even then the values are approximations. Only the device you're programming knows the exact storage size of C's standard variables.

Listing 18-1 uses the `sizeof` operator to determine how much storage each C language variable type occupies in memory.

Listing 18-1: How Big Is a Variable?

```
#include <stdio.h>

int main()
{
    char c = 'c';
    int i = 123;
    float f = 98.6;
    double d = 6.022E23;

    printf("char\t%u\n",sizeof(c));
    printf("int\t%u\n",sizeof(i));
    printf("float\t%u\n",sizeof(f));
    printf("double\t%u\n",sizeof(d));
    return(0);
}
```

Exercise 18-1: Type the source code from Listing 18-1 into your editor. Build and run to see the size of each variable type.

Here's the output I see:

```
char    1
int     4
float   4
double  8
```

The `sizeof` keyword isn't a function. It's more of an operator. Its argument is a variable name. The value that's returned is of the C language variable type known as `size_t`. Without getting into a long, boring discussion, the `size_t` variable is a *typedef* of another variable type, such as an `unsigned int` on a PC or a `long unsigned int` on other computer systems. The bottom line is that the size indicates the number of bytes used to store that variable.

Arrays are also variables in C, and `sizeof` works on them as shown in Listing 18-2.

Listing 18-2: How Big Is an Array?

```c
#include <stdio.h>

int main()
{
    char string[] = "Does this string make me look fat?";

    printf("The string \"%s\" has a size of %u.\n",
            string,sizeof(string));
    return(0);
}
```

Exercise 18-2: Type the source code from Listing 18-2. Build and run it to see how much storage the `char` array occupies.

Exercise 18-3: Edit your source code from Exercise 18-2, adding the `strlen()` function to compare its result on the array with the `sizeof` operator's result.

If the values returned by `strlen()` and `sizeof` differ, can you explain the difference?

Okay, I'll explain: When you create an array, the program allocates space in memory to hold the array's values. The allocation is based on the size of each element in the array. So a `char` array of 35 items (including the `\0`, or NULL) occupies 35 bytes of storage, but the length of the string is still only 34 characters (bytes).

Exercise 18-4: Edit the source code from Exercise 18-2 again, this time creating an `int` array with five elements. The array need not be assigned any values, nor does it need to be displayed. Build and run.

Can you explain the output? If not, review the output from Exercise 18-1. Try to figure out what's happening. See Listing 18-3.

Listing 18-3: How Large Is a Structure?

```
#include <stdio.h>

int main()
{
    struct robot {
        int alive;
        char name[5];
        int xpos;
        int ypos;
        int strength;
    };

    printf("The evil robot struct size is %u\n",
            sizeof(struct robot));
    return(0);
}
```

Exercise 18-5: Start a new project using the code shown in Listing 18-3. Build and run to determine the size of the structure variable.

The `sizeof` operator works on all variable types, but for a structure, specify the structure itself. Use the keyword `struct` followed by the structure's name, as shown in Line 14. Avoid using a structure variable.

The size of the structure is determined by adding up the storage requirement for each of its elements. You might assume, given the size output from Exercise 18-5, that four `int` variables plus five `char` variables would give you 21: $4 \times 4 + 1 \times 5$. But it doesn't work that way.

On my screen I see this output:

```
The evil robot struct size is 24
```

The reason you see a value other than `21` is that the program aligns variables in memory. It doesn't stack them up, one after another. If I were to guess, I would say that 3 extra bytes are padded to the end of the `name` array to keep it aligned with an 8-byte offset in memory. Figure 18-1 illustrates what's going on.

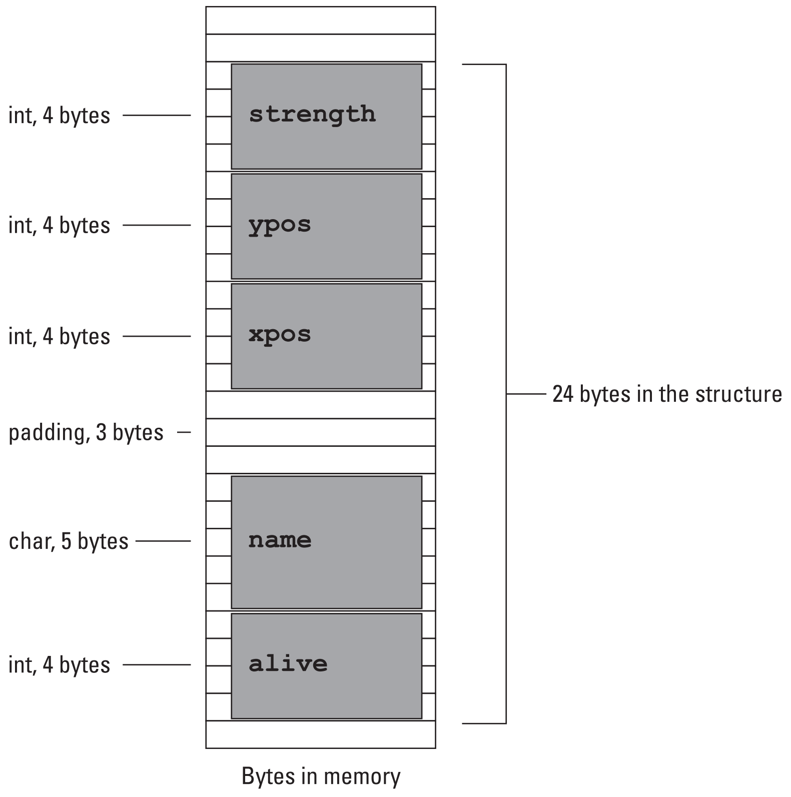

Figure 18-1: How a structure fits in memory.

- The `sizeof` operator returns the size of a C language variable. It includes the size of a structure.
- You cannot use `sizeof` to determine the size of your program or the size of anything other than a variable.
- When you use `sizeof` on a structure variable, you get the size of that variable, which can be less than the structure's declared size. A problem can occur when writing structures to a file if you use the variable's size rather than the structure's defined size. See Chapter 22.
- The 8-byte offset used to align variables in memory keeps the CPU happy. The processor is much more efficient at reading memory aligned to those 8-byte offsets.
- The values returned by `sizeof` are most likely bytes, as in 8 bits of storage. That size is an assumption: Just about every electronic gizmo today uses an 8-bit byte as the standard storage unit. That doesn't mean you won't find a gizmo with a 7-bit byte or even a 12-bit byte. Just treat the values returned by `sizeof` as a "unit" and you'll be fine.

Checking a variable's location

A variable's type and size are uncovered first by declaring that variable as a specific type, but also by using the `sizeof` keyword. The second description of a variable, its contents, can be gleaned by reading the variable's value using the appropriate C language function.

The third description of a variable is its location in memory. You gather this information by using the `&` operator and the `%p` placeholder, as shown in Listing 18-4.

Listing 18-4: O Variable, Wherefore Art Thou?

```
#include <stdio.h>

int main()
{
    char c = 'c';
    int i = 123;
    float f = 98.6;
    double d = 6.022E23;

    printf("Address of 'c' %p\n",&c);
    printf("Address of 'i' %p\n",&i);
    printf("Address of 'f' %p\n",&f);
    printf("Address of 'd' %p\n",&d);
    return(0);
}
```

When the `&` operator prefixes a variable, it returns a value representing the variable's *address,* or its location in memory. That value is expressed in hexadecimal. To view that value, the %p conversion character is used, as shown in Listing 18-4.

Exercise 18-6: Type the source code from Listing 18-4 into your editor. Build and run.

The results produced by the program generated from Exercise 18-6 are unique, not only for each computer but also, potentially, for each time the program is run. Here's what I see:

```
Address of 'c' 0x7fff5fbff8ff
Address of 'i' 0x7fff5fbff8f8
Address of 'f' 0x7fff5fbff8f4
Address of 'd' 0x7fff5fbff8e8
```

Variable *c* is stored in memory at location 0x7fff5fbff8ff — that's decimal location 140,734,799,804,671. Both values are trivial, of course; the computer keeps track of the memory locations, which is just fine by me. Figure 18-2 offers a memory map of the results just shown.

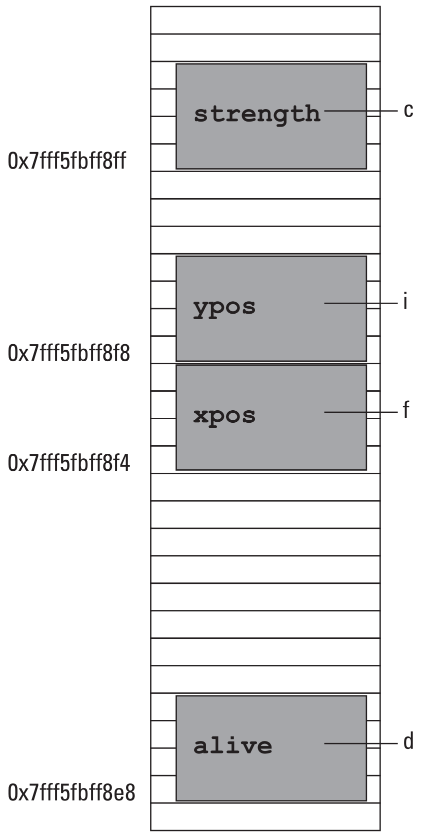

Figure 18-2: Variable locations in memory.

I can offer no explanation why my computer chose to place the `int` variables where it did, but Figure 18-2 illustrates how those addresses map out in memory.

Individual array elements have memory locations as well, as shown in Listing 18-5 on Line 10. The `&` operator prefixes the specific element variable, coughing up an address. The `%p` conversion character in the `printf()` function displays the address.

Listing 18-5: Memory Locations in an Array

```
#include <stdio.h>

int main()
{
    char hello[] = "Hello!";
    int i = 0;

    while(hello[i])
    {
        printf("%c at %p\n",hello[i],&hello[i]);
        i++;
    }
    return(0);
}
```

Exercise 18-7: Create a new project by using the source code shown in Listing 18-5. Build and run.

Again, memory location output is unique on each computer. Here's what I see:

```
H at 0x7fff5fbff8f0
e at 0x7fff5fbff8f1
l at 0x7fff5fbff8f2
l at 0x7fff5fbff8f3
o at 0x7fff5fbff8f4
! at 0x7fff5fbff8f5
```

Unlike the example from Exercise 18-6, the addresses generated by Exercise 18-7 are contiguous in memory, one byte after another.

Exercise 18-8: Code a program to display five values in an `int` array along with each element's memory address. You can use Listing 18-5 to inspire you, although a `for` loop might be easier to code.

- By the way, the `&` memory location operator should be familiar to you. It's used by the `scanf()` function, which requires a variable's address, not the variable itself. That's because `scanf()` places a value at a memory location directly. How? By using pointers, of course!
- The `&` operator is also the bitwise AND operator; however, the compiler is smart enough to tell when `&` prefixes a variable and when `&` is part of a binary math equation.

Reviewing variable storage info

To summarize this section, variables in C have a name, type, value, and location.

- The variable's type is closely tied to the variable's size in memory, which is obtained by using the `sizeof` operator.
- A variable's value is set or used directly in the code.
- The variable's location is shown courtesy of the `&` operator and the `%p` conversion character.

When you have a basic understanding of each of the elements in a variable, you're ready to tackle the hideously complex topic of pointers.

The Hideously Complex Topic of Pointers

Memorize this sentence:

A pointer is a variable that contains a memory location.

Or maybe this story will help:

Once upon a time, a pointer variable met a college student enrolled in a C programming course. The student asked, "What do you point at?" The variable replied, "Nothing! But I contain a memory location." And the freshman was severely satisfied.

You must accept the insanity of the pointer before moving on. True, though you can get at a variable's memory location, or *address,* by using the `&` operator, the pointer is a far more powerful beast.

Introducing the pointer

A pointer is a type of variable. Like other variables, it must be declared in the code. Further, it must be initialized before it's used. That last part is really important, but first the declaration has this format:

```
type *name;
```

As when you declare any variable, the *type* identifies the pointer as a `char`, `int`, `float`, and so on. The *`name`* is the pointer variable's name, which must be unique, just like any other variable name. The asterisk identifies the variable as a pointer, not as a regular variable.

The following line declares a `char` pointer, *`sidekick`*:

```
char *sidekick;
```

And this line creates a `double` pointer:

```
double *rainbow;
```

To initialize a pointer, you must set its value, just like any other variable. The big difference is that a pointer is initialized to the memory location. That location isn't a random spot in memory, but rather the address of another variable within the program. For example:

```
sidekick = &lead;
```

The preceding statement initializes the *`sidekick`* variable to the address of the *`lead`* variable. Both variables are `char` types; if not, the compiler would growl like a wet cat. After that statement is executed, the *`sidekick`* pointer contains the address of the *`lead`* variable.

If you're reading this text and just nodding your head without understanding anything, good! That means it's time for an example.

I've festooned the source code in Listing 18-6 with comments to help describe two crucial lines. The program really doesn't do anything other than prove that the pointer *`sidekick`* contains the address, or memory location, of variable *`lead`*.

Listing 18-6: An Example

```
#include <stdio.h>

int main()
{
    char lead;
    char *sidekick;

    lead = 'A';         /* initialize char variable */
    sidekick = &lead;   /* initialize pointer IMPORTANT! */

    printf("About variable 'lead':\n");
    printf("Size\t\t%ld\n",sizeof(lead));
    printf("Contents\t%c\n",lead);
    printf("Location\t%p\n",&lead);
    printf("And variable 'sidekick':\n");
    printf("Contents\t%p\n",sidekick);

    return(0);
}
```

Other things to note: Line 12 uses two tab escape sequences to line up the output. Also, don't forget the `&` in Line 14, which fetches the variable's address.

Exercise 18-9: Type the source code from Listing 18-6 into your editor. Build and run.

Here's the output I see on my screen:

```
About variable 'lead':
Size            1
Contents        A
Location        0x7fff5fbff8ff
And variable 'sidekick':
Contents        0x7fff5fbff8ff
```

The addresses (in the example) are unique for each system, but the key thing to note is that the contents of pointer *`sidekick`* are equal to the memory location of variable *`lead`*. That's because of the assigning, or initialization, that takes place on Line 9 in the code.

It would be pointless for a pointer to merely contain a memory address. The pointer can also peek into that address and determine the value that's stored there. To make that happen, the `*` operator is prefixed to the pointer's variable name.

Exercise 18-10: Modify your source code from Exercise 18-9 by adding the following statement after Line 16:

```
printf("Peek value\t%c\n",*sidekick);
```

Build and run. Here's the output I see as output:

```
About variable 'lead':
Size            1
Contents        A
Location        0x7fff5fbff8ff
And variable 'sidekick':
Contents        0x7fff5fbff8ff
Peek value      A
```

When you specify the `*` (asterisk) before an initialized pointer variable's name, the results are the contents of the address. The value is interpreted based on the type of pointer. In this example, *`*sidekick`* represents the `char` value stored at a memory location kept in the *`sidekick`* variable, which is really the same as the memory location variable *`lead`*.

To put it another way:

- A pointer variable contains a memory location.
- The *pointer* variable peeks into the value stored at that memory location.

Working with pointers

The pointer's power comes from both its split personality as well as from its ability to change values, such as a variable.

In Listing 18-7, three char variables are declared at Line 5 and initialized all on Line 8. (I stacked them up on a single line so that the listing wouldn't get too long.) A char pointer is created at Line 6.

Listing 18-7: More Pointer Fun

```
#include <stdio.h>

int main()
{
    char a,b,c;
    char *p;

    a = 'A'; b = 'B'; c = 'C';

    printf("Know your ");
    p = &a;                 // Initialize
    putchar(*p);            //  Use
    p = &b;                 // Initialize
    putchar(*p);            //  Use
    p = &c;                 // Initialize
    putchar(*p);            //  Use
    printf("s\n");

    return(0);
}
```

Lines 11 and 12 set up the basic operation in the code: First, pointer *p* is initialized to the address of a char variable. Second, the * (asterisk) is used to peek at the value stored at that address. The **p* variable represents that value as a char inside the putchar() function. That operation is then repeated for char variables *b* and *c*.

Exercise 18-11: Create a new project by using the source code from Listing 18-7. Build and run.

Figure 18-3 attempts to illustrate the behavior of pointer variable `p` as the code runs.

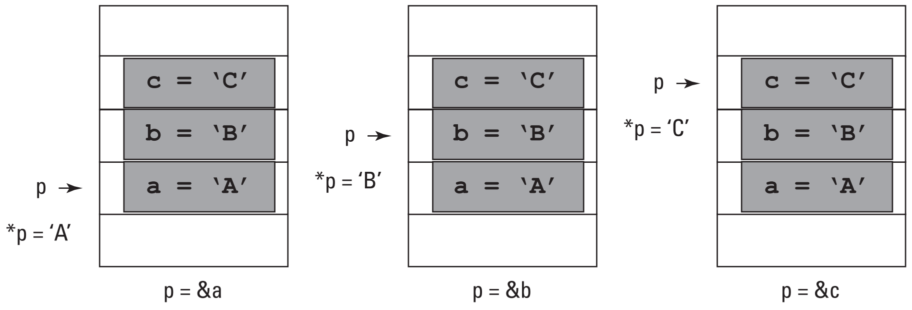

Figure 18-3: Using a pointer to read values.

Exercise 18-12: Write a program that declares both an `int` variable and an `int` pointer variable. Use the pointer variable to display the value stored by the `int` variable.

The `*pointer` operator works both ways. Just as you can grab a variable's value, as shown in Listing 18-7, you can also set a variable's value. Refer to Listing 18-8.

Listing 18-8: Assigning Values by Using a Pointer

```
#include <stdio.h>

int main()
{
    char a,b,c;
    char *p;

    p = &a;
    *p = 'A';
    p = &b;
    *p = 'B';
    p = &c;
    *p = 'C';
    printf("Know your %c%c%cs\n",a,b,c);
    return(0);
}
```

Line 5 in Listing 18-8 declares three `char` variables. These variables are never directly assigned values anywhere in the code. The *`p`* variable, however, is initialized thrice (Lines 8, 10, and 12) to the memory locations of variables *`a`*, *`b`*, and *`c`*. Then the *`*p`* variable assigns values to those variables (Lines 9, 11, and 13.) The result is displayed by `printf()` at Line 14.

Exercise 18-13: Copy the source code from Listing 18-8 into your editor. Build and run the program.

Exercise 18-14: Write code that declares an `int` variable and a `float` variable. Use corresponding pointer variables to assign values to those variables. Display the results by using the `int` and `float` variables (not the pointer variables).